CLIPS Documentation

Three documents are provided with CLIPS.

- The *CLIPS* Reference Manual which is split into the following parts:
  - *Volume I - The* Basic Programming Guide, which provides the definitive description of CLIPS syntax and examples of usage.
  - *Volume II - The* Advanced Programming Guide, which provides detailed discussions of the more sophisticated features in CLIPS and is intended for people with extensive programming experience who are using CLIPS for advanced applications.
  - *Volume III - The* Interfaces Guide, which provides information on machine-specific interfaces. *CLIPSWIG.DOC e l'estratto di questo manuale per Windows 3.1*
- The *CLIPS* Users Guide which provides an introduction to CLIPS and is intended for people with little or no expert system experience.
  - *Volume I - Rules*, which provides an introduction to rule-based programming using CLIPS.
  - *Volume II - Objects*, which provides an introduction to object-oriented programming using COOL.

The *CLIPS Architecture Manual* which provides a detailed description of the CLIPS software architecture. This manual describes each module of CLIPS in terms of functionality and purpose. It is intended for people with extensive programming experience who are interested in modifying CLIPS or who want to gain a deeper understanding of how CLIPS works.

COSA E' CLIPS ?

E' un tool di sviluppo per sistemi esperti sviluppato *da Software Tecnology Branch (STB) NASA/Lyndon Space Center*
(info news group comp.ai.shells)

CLIPS E' PENSATO PER FACILTARE LO SVILUPPO DI SOFTWARE BASATO SULLA RAPPRESENTAZIONE DELLA CONSCENZA O ESPERIENZA UMANA

CLIPS NON E' UN LINGUAGGIO INTELLIGENTE !

CLIPS NON E' UN COMPILATORE !

CLIPS SI COMPONE DI:

- **FATTI: descrivono la realta'**

```
(assert (semaforo rosso))   ;fatto "semaforo" valore rosso
```

- **REGOLE: sono la parte elementare per esprimere la conoscenza basata sull'esperienza e lavorano sui fatti**

```
(defrule REGOLA1 "esempio sintassi di una regola"
   (Animale Cane)              ;ipotesi (LHS:Left Hand Side)
=>
   (assert (Verso Bau))    ;tesi (RHS:Right Hand Side)
)
```

- **FUNZIONI: rappresentano la conoscenza algoritmica**

```
(deffunction IPOTENUSA (?a ?b) "esempio di una funzione"
   (sqrt (+ (* ?a ?a) (* ?b ?b))
) ;ritorna il risultato dell'ultima espressione cioè sqrt
```

Un programma scritto in CLIPS e' composto da regole e fatti[*] .

Il MOTORE INFERENZIALE decide quali regole devono essere eseguite e quando

Un sistema esperto basato su regole e' un programma *data-driven* dove i fatti sono i dati che attraverso il motore inferenziale stimolano l'esecuzione delle regole.

[*] **In realta vi sono anche moduli, oggetti e funzioni. Tuttavia questi sono di complemento al linguaggio ed hanno lo scopo di renderlo piu' funzionale**

In un linguaggio procedurale come il C, Pascal, Basic ecc.. l'esecuzione puo' procedere anche senza dati.

CLIPS v.6.0 per Windows 3.1

ISTRUZIONI PER L'INSTALLAZIONE:

1- Copiare il contenuto dei 3 dischetti in una directory temporanea

2- Scompattare i file .zip nella directory temporanea

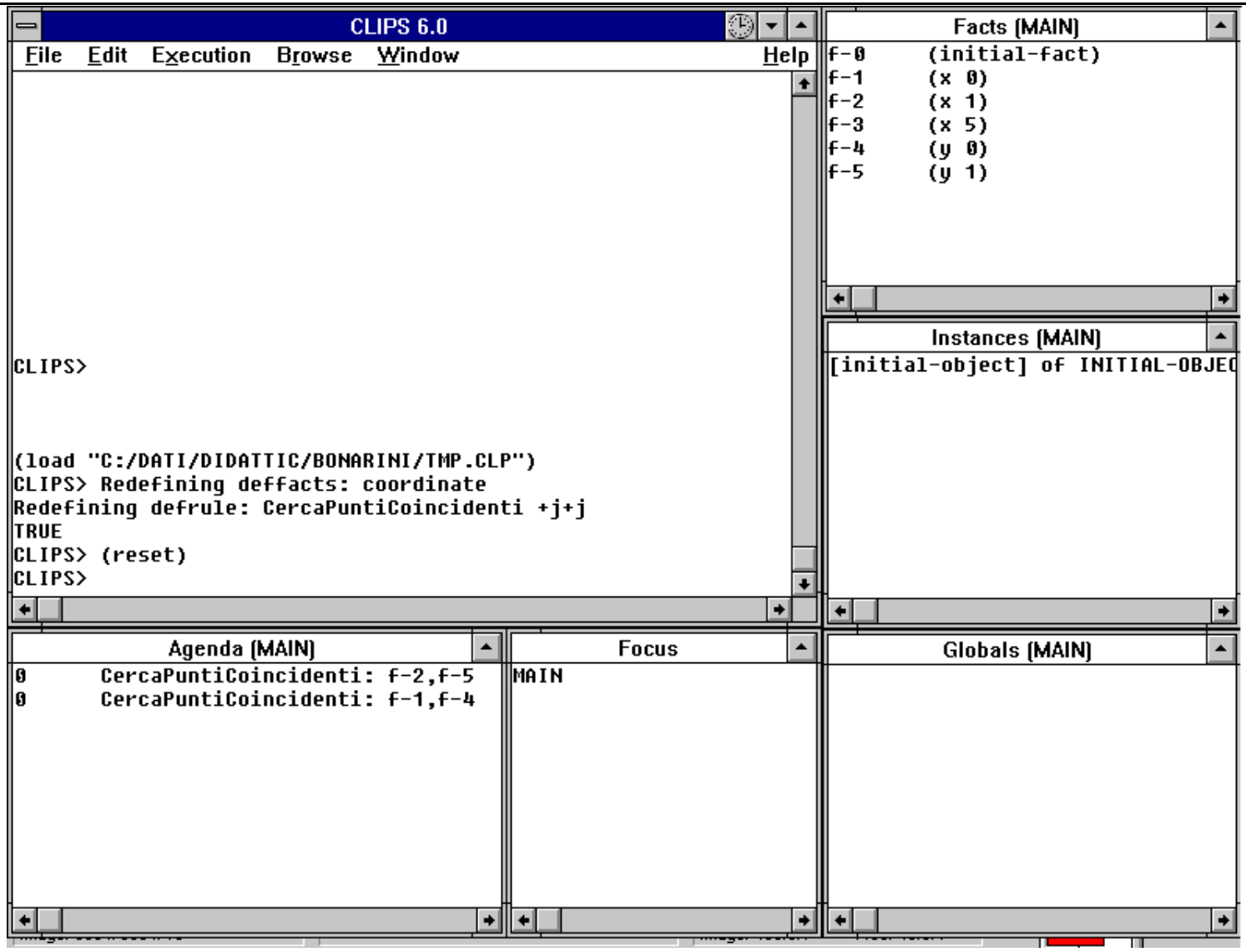
3*- Lanciare il file di installazione *DOSINST.EXE
per fare una installazione completa servono :

- **2.5MB per CLIPS**
- **100KB per gli esempi**
- **3.3MB per la documentazione di cui 1.3MB per i file USRGUIDE.DOC, CLIPSWIG.DOC che e' consigliabile avere**

4- Copiare CLIPSWIG.DOC in \CLIPS\DOUCUMENT

5- Entrare in Windows e lanciare il file CLIPSWIN.EXE

CLIPSWIG.DOC e ICSE.DOC(queste dispense) sono disponibili all'indirizzo http://www.elet.polimi.it/people/contini/icse


CLIPS 6.0
File Edit Execution Browse Window Help
CLIPS>
(load "C:/DATI/DIDATTIC/BONARINI/TMP.CLP")
CLIPS> Redefining deffacts: coordinate
Redefining defrule: CercaPuntiCoincidenti +j+j
TRUE
CLIPS> (reset)
CLIPS>
Facts (MAIN)
f-0 (initial-fact)
f-1 (x 0)
f-2 (x 1)
f-3 (x 5)
f-4 (y 0)
f-5 (y 1)
Instances (MAIN)
[initial-object] of INITIAL-OBJEC
Agenda (MAIN)
0 CercaPuntiCoincidenti: f-2,f-5
0 CercaPuntiCoincidenti: f-1,f-4
Focus
MAIN
Globals (MAIN)

Gli elementi dell'interfacia di CLIPS 6.0 sono:

- ***La finestra principale*** **presenta il prompt di clips `CLIPS>` all'operatore. E' pronto per accettare dei comandi in linguaggio clips**

- ***Fatcs*** **presenta l'elenco dei fatti in memoria per il modulo corrente. I fatti sono numerati in ordine di inserimento `f-1 f-2,..,f-n`**

- ***Agenda*** **presenta l'elenco delle regole attivate dai fatti per il modulo corrente. Per ogni regola attivata viene indicato l'elenco dei fatti che la ha attivata e la *salience* della regola.**

- ***Focus*** **elenco dei moduli. Il primo della lista e' il modulo corrente.**

- ***Global* e *Instaces* visualizzano l'elenco delle variabili globali e degli oggetti instanziati**

Fare riferimento a CLIPSWIG.DOC per un manuale dettagliato del software

L'AGENDA

- **L'agenda e' una lista di regole le cui condizioni sono soddisfatte dai fatti nella base dei fatti e non sono state ancora eseguite**
- **Ogni modulo ha una propria agenda**
- **L'agenda viene gestita come uno stack: la regola in testa allo stack e' la prima ad essere eseguita**
- **Quando una nuova regola, per la presenza di un nuovo fatto, viene attivata, viene inserita nella agenda secondo la strategia impostata.**

- **Ad una regola puo' essere associato il concetto di *salience* che rappresenta l'importanza di una regola. Il range di salience e' ±10000 mentre per default ogni regola ha salience 0**

CONFLICT RESOLUTION STRATEGIES

ESISTONO 7 DIFFERENTI STRATEGIE PER LA GESTIONE DELL'AGENDA

- ***DEPTH (default):*** **Le nuove regole vengono poste al di SOPRA delle regole con la stessa salience. Se due regole vengono attivate allo stesso momento allora la loro posizione relativa nell'agenda sara' arbitraria**

- ***BREADTH:*** **Le nuove regole vengono poste al di SOTTO delle regole con la stessa salience. Se due regole vengono attivate allo stesso momento allora la loro posizione relativa nell'agenda sara' arbitraria**

- ***SIMPLICITY*: tra le regole con la stessa *salience una* nuova regola viene messa sopra tutte le regole con pari o maggiore *SPECIFICITA'.* La specificita' di una regola e' il numero fatti necessari a far scattare la regola. (le funzioni and,not e or non si considerano nel calcolo della specificita', ma i loro argomenti si; le chiamate a funzioni non aumentano la specificita di una regola)**

ESEMPIO

```
(defrule example     ;regola con specificita' 5
   (item ?x ?y ?x)
   (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
   =>)
```

- ***COMPLEXITY*: tra le regole con la stessa salience una nuova regola viene messa sopra tutte le regole con pari o minore specificita'.**

- ***RANDOM*: Ogni attivazione viene assegnata ad un numero casuale che determina la posizione nell'agenda tra le regole con pari *salience***

- ***LEX*: per ogni fatto viene impostato un *time tag* che misura quanto il fatto e' recente rispetto agli altri. Una regola attivata con fatti piu' recenti e' posta al di SOPRA di regole attivate da fatti piu' vecchi, tra le regole con pari *salience***

- ***MEA*: per ogni fatto viene impostato un *time tag* che misura quanto il fatto e' recente rispetto agli altri. Una regola attivata con fatti piu' recenti e' posta al di SOTTO di regole attivate da fatti piu' vecchi, tra le regole con pari *salience***

NOTAZIONE

CLIPS ha una notazione simile al LISP in cui

ogni funzione e' chiamata con parentesi

```
CLIPS> (exit)                    ;fine del programma
```

le operazioni artimetiche sono in notazione inversa

```
CLIPS>(+ 3 4)                    ;=>3+4

CLIPS>(* (- a b) (+ a b))        ;=>(a-b)*(a+b)
CLIPS>(/ (- (sqrt (- (* b b) (* 4 a c))) b)(* 2 a))
                                 ;=>(-b+sqrt(b^2 - 4ac))/2a
```

VARIABILI

VARIABILI CHE CONTEGONO VALORI DI UN FATTO

```
(defrule Semaforo
    (semaforo-e ?colore)
=>
    (printout t crlf “Il semaforo è “ ?colore crlf)
)
```

VARIABILI CHE CONTENGONO L’INDIRIZZO DI UN FATTO

```
(defrule Matrimonio
    ?il-celibe <- (e-celibe Andrea)
=>
    (printout t crlf “Andrea ora è sposato”)
    (retract ?il-celibe)
)
```

ESEMPIO:

```
(deffacts ListaCelibi
    (celibe Andrea)
    (celibe Giovanni)
    (celibe Michele)
)

(defrule Matrimono “Questa regola sposa ed elimina tutti i
                        celibi”
    ?celibe <- (celibe ?nome)
=>
    (printout t crlf ?nome “ ora è sposato” crlf)
    (assert (sposato ?nome))
    (retract ?celibe)
)
```

NOTA: una regola scatta su TUTTI i fatti che corrispondono al suo pattern posto nella LHS, quindi questa regola scatta 3 volte !

VARIABILI ANONIME: sono quelle variabili di cui non viene indicato il nome. Sono utili quando è necessario che esista un elemento nella lista del pattern di cui non interessa il contenuto.

ESEMPIO:

```
(deffacts ListaCelibi
   (celibe Andrea)
   (celibe Giovanni)
   (celibe Michele)
   (celibe)
)

(defrule EliminaCelibi “Questa regola elimina tutti i celibi”
   ?celibe <- (celibe ?)
=>
   (retract ?celibe)
)
```

QUESTA REGOLA VIENE ATTIVATA SOLO SUI PRIMI TRE FATTI

E’ POSSIBILE USARE PIU’ DI UNA VARIABILE PER IL PATTERN MATCHING.

ESEMPIO:

```
(deffacts ListaCelibi
    (celibe Andrea biondo)
    (celibe Giovanni castano)
    (celibe Michele)
)
```

se volessi “eliminare” tutti i celibi indipendentemente dal colore dei capelli, potrei scrivere:

```
(defrule EliminaCelibi1
    ?celibe <- (celibe ?)
=>
    (retract ?celibe))

(defrule EliminaCelibi2
    ?celibe <- (celibe ? ?)
=>
    (retract ?celibe))
```

C'E' UN MODO PIU' EFFICIENTE PER RISOLVERE IL PROBLEMA ?

IN CLIPS E' POSSIBILE USARE LA WILDCARD $? PER INDICARE *ZERO O PIU' CAMPI*. IN QUESTO MODO POSSIAMO SCRIVERE:

```
(deffacts ListaCelibi
   (celibe Andrea biondo)
   (celibe Giovanni castano)
   (celibe Michele)
)
(defrule EliminaCelibi
   ?celibe <- (celibe $?)
=>
   (retract ?celibe)
)
```

questa regola viene attivata su tutti i fatti *celibe*

UNA WILDCARDS PUO' ESSERE ANCHE ASSOCIATA AD UN NOME SIMBOLICO COSI' COME UNA VARIABILE *$?nome-wildcards*

```
(deffacts ListaCelibi
   (celibe Andrea biondo)
   (celibe Giovanni castano)
   (celibe Michele)
)

(defrule EliminaCelibi
   ?celibe <- (celibe  $?nome)
=>
   (printout t crlf "Elimino " ?nome)
   (retract ?celibe)
)
```

NOTA: la variabile multicampo *$?nome* nella parte RHS diventa *?nome*

LE VARIABILI MULTICAMPO POSSONO ESSERE USARE PER CERCARE UN MATCH CON UN FATTO CHE CONTIENE IN UNO DEI SUOI CAMPI UN VALORE SPECIFICO

si supponga di avere la seguente lista:

```
(deffacts ListaCelibi
   (celibe Andrea ricco biondo)
   (celibe Giovanni castano intelligente povero)
   (celibe Michele intelligente)
)
```

per estrarre tutti gli intelligenti si puo' scrivere la seguente regola

```
(defrule CelibiIntelligenti
   (celibe ?nome $? intelligente $?)
=>
   (printout t ?nome " è intelligente !" crlf)
)
```

LE VARIABILI MULTICAMPO IN MOLTI CASI POSSONO ESSERE ESSENZIALI PER IL PATTERN MATCHING MA IL LORO USO E' INEFFICIENTE A CAUSA DELLA QUANTITA' DI MEMORIA NECESSARIA PER LA LORO GESTIONE

COME REGOLA DI BUONA IMPLEMENTAZIONE SI CONSIGLI A DI USARE $? SOLO QUANDO NON E' NOTO IL NUMERO DI CAMPI DI UN FATTO.

NON USARE $? SOLO PER COMODITA !!

L'USO DELLE VARIABILI PER IL PATTERN MATCHING GODE DI UNA IMPORTANTE ED UTILE PROPRIETA':

ALLA PRIMA OCCORENZA, UNA VARIABILE VIENE ASSEGNATA E MANTIENE IL SUO VALORE SOLO ALL'INTERNO DELLA REGOLA.

TUTTE LE SUCCESSIVE OCCORENZE DI UNA VARIABILE FORNISCONO IL VALORE PRECEDENTEMENTE ASSEGNATOGLI.

QUESTA PROPRIETA' CI PERMETTE AD ESEMPIO DI

```
(deffacts coordinate
   (x 0)
   (x 1)
   (x 5)
   (y 0)
   (y 1)
)
(defrule CercaPuntiBisettrice
   (x ?valore)
   (y ?valore)
=>
   (printout t "Trovata la coppia x,y con valore " ?valore
crlf)
)
```

Questa regola viene attivata solo per quelle coppie di fatti x,y i cui campi che seguono hanno lo stesso valore.

In pratica con *(x ?valore) ?valore* viene assegnato con *(y ?valore) ?valore* viene solo letto

UN ALTRO ESEMPIO

```
(deffacts ListaAttributi
    (biondo Marco)
    (biondo Andrea)
    (moro Michele)
    (castano Francesco)
    (intelligente Michele)
    (normale Andrea)
    (non-intelligente Marco)
    (intelligente Francesco)
)

(defrule MoriIntelligenti
    (moro ?nome)
    (intelligente ?nome)
=>
    (printout t ?nome " è moro ed intelligente" crlf)
)
```

ESEMPIO: ordinare una lista di fatti secondo gli attributi

```
(deffacts ListaScapoli
   (lista Giovanni Mario Andrea)
)

(deffacts AttributiScapoli
   (occhi scuri Andrea)
   (occhi scuri molto Mario)
   (occhi chiari Giovanni)
   (carattere simpatico Andrea)
   (carattere silenzioso Mario)
   (carattere silenzioso Giovanni)
)

(defrule ScapoloIdeale
   (occhi scuri ?nome)
   (carattere simpatico ?nome)
=>
   (printout t ?nome " è lo scapolo ideale " crlf)
   (assert (inTesta ?nome))
```

```
)
```

```
(defrule InTesta
    ?perLaTesta <- (inTesta ?chi)
    ?vecchiaLista <- ( lista $?testa ?chi $?coda)
=>
    (retract ?perLaTesta ?vecchiaLista)
    (assert (lista ?chi ?testa ?coda))
    (assert (cambiaLista si))  ;per attivare StampaLista
)

(defrule StampaLista
    ?cambiaLista <- (cambiaLista si)
    (lista $?lista)
=>
    (retract ?cambiaLista)
    (printout t "La lista è: " ?lista crlf)
)
```

DEFTEMPLATE

- **Un template e' simile alla struttura di una tabella**

- ***deftemplate* definisce, in un unico pattern, un gruppo di campi che sono in relazione tra loro**

- **Un template e' una lista di slot . Uno slot e' composto dal nome seguito dal suo valore/i**
  - **Un *single-slot* contiene esattamente un valore**
  - **Un *multi-slot* puo' contenere zero o piu' valori**

uno slot è un campo, l'insieme degi slot e' un record, l'insieme dei record e' la tabella

ESEMPIO:

```
(deftemplate Film
   (slot titolo   (type STRING)(default ?DERIVE))
   (slot regista (type STRING)(default ?DERIVE))
   (slot genere   (type SYMBOL)(default ?DERIVE))
   (slot anno    (type NUMBER))
)
```

In questo esempio troviamo il nome del template *Prospetto* la lista degli attributi chiamati campi. Per ogni campo e' specificato il tipo di dato che deve essere contenuto, come anche il vaolre di default. I tre campi sono tutti *single-slot*

- **I valori di default vengono inseriti da CLIPS al momento dell'assert del template**

- **?DERIVE selezione al valore di default appropriato per il tipo indicato**

- **NUMBER puo’ essere sia INTEGER che FLOAT**

ESEMPIO: Inserimento di un record:

```
(assert(Film (anno 1995) (titolo “Panther”)))

..
f-0 (Film (titolo “”)(regista “”)(genere nil)(annno 0))
f-1 (Film (titolo “” )(regista “”)(genere nil)(anno 1995))
```

- **I campi non inseriti esplicitamente vengono inseriti con il valore di default**
- **L’ordine con cui si indicano i campi al momento dell’inserimento non e’ importante: *quello che conta e’ il nome del campo***

ESEMPIO: scelta di un film

```
(deffacts ListaFilm
   (Film (genere commedia)(titolo “Amici miei III”))
   (Film (anno 1995)(titolo “Panther”)(regista “Van
Peebles”)(genere metropolitano))
   (Film (anno 1990) (titolo “Brazil”) (genere futuristico))
   (Film (titolo “Brancaleone”)(regista “Gasmann”)(genere
commedia))
)

(defrule SceltaCommedie
   (Film (genere commedia)(titolo ?titolo))
=>
   (printout t “Ho trovato: “ ?titolo crlf)
)
```

i template multislot possono aiutare a generare delle liste piu' dettagliate senza complicare la fase di pattern matchin nelle regole.

Si immagini di voler inserire per ogni film anche la lista degli attori principali, e poi scegliere tutti i film in cui compare un determinato attore. Si proceda come segue:

```
(deftemplate Film
   (slot titolo   (type STRING)(default ?DERIVE))
   (slot regista (type STRING)(default ?DERIVE))
   (multislot attori (type STRING))
   (slot genere   (type SYMBOL)(default ?DERIVE))
   (slot anno    (type NUMBER))
)
```

il campo attori e' un *multi-slot* di tipo stringa: puo' contenere dunque piu' di una stringa.

```
(deffacts ListaFilm
   (Film (genere commedia)(titolo “Amici miei III”)
      (attori “Tognazzi”))
   (Film (genere commedia)(titolo “Marcia su Roma”)
      (attori “Tognazzi” “Gasmann”))
   (Film (anno 1995)(titolo “Panther”)(regista “Van
Peebles”)(genere metropolitano))
   (Film (titolo “Brancaleone”)(regista “Monicelli”)
      (genere commedia)(attori “Gasmann”))
)

(defrule SceltaGasmann
   (Film (attori $? Gasmann $?)(titolo ?titolo))
=>
   (printout t “Ho trovato: “ ?titolo crlf)
)
```

FUNZIONI

In CLIPS sono è possibile usere dei *connettivi* logici sul pattern matching.

Si consideri il problema di guidare un robot di fronte ad un semaforo:
luce -rosso => stop
luce -verde => avanza
luce -giallo => attenzione
luce -lampeggiaGiallo => attenzione
pedoni-avanti => avanza
pedoni-alt => stop

come fare per far avanzare il robot solo sel il semaforo e' verde ?

CONNETTIVO NOT (~)

```
(defrule Stop
   (luce ~verde) ;semaforo ~ (non e') verde
=>
   (printout t "Stop !!" crlf)
)
```

in questo caso si vincola l'avanzamento del robot solo quando e' accesa la luce verde al semaforo.

CONNETTIVO OR (|)

```
 (defrule Attenzione
   (luce giallo|lampeggiaGiallo)
=>
   (printout t "Attenzione !!" crlf)
)
```

Questa regola scatta se il semaforo e' giallo oppure e' lampeggiante giallo

CONNETTIVO AND (&)

```
(defrule NonGialloNonRosso
    (luce ?colore&~rosso&~giallo)
=>
    (printout t “Avanza ! la luce è “ ?colore crlf)
)
```

In questa regola viene cercato un pattern con due elementi, il primo deve essere *luce* il secondo valore viene messo in ?colore.
Per essere verificato il pattern matching deve essere

```
(?colore AND NOT rosso) AND (?colore AND NOT giallo)
```

Nel caso conoscere il colore non e’ necessario allora si poteva scivere:

```
(defrule NonGialloNonRosso
    (luce ~rosso&~giallo)
=>
    (printout t “Avanza !“ ?colore crlf))
```

CONNETTIVO EQUAL (=)

è usato per indicare a CLIPS di valutare l'espressione che segue per usarla nel pattern matching.
Si supponga di dover scegliere la risposta corretta in una lista di risposte possibile alla domanda:
Scrivere una regola che risponde al quiz seguente:

```
(deffacts Quiz
   (domanda cateti 3 4)
   (risposta A 2.0)
   (risposta B 5.0)
   (risposta C 7.0)
)
(defrule RispostaAlQuiz
   (domanda cateti ?x ?y)
   (risposta ?ID ?valore& = (sqrt(+ (** ?x 2) (** ?y 2))))
=>
   (printout t "La risposta è la " ?ID " con " ?valore crlf)
)
```

BIND

La funzione *bind* è usata per assegnare una variabile nel RHS di una regola E' l'analogo dell'assegnamento di una variabile nel LHS con il pattern matching.

```
(defrule Somma
    (numeri ?x ?y)
=>
    (bind ?somma (+ ?x ?y))
    (assert (somma ?somma))
    (printout "La somma vale: " ?somma crlf)
)
```

DEFFUNCTION

Come normali linguaggi di programmazione CLIPS permette di definire delle funzioni con parametri e valore di ritorno.

L'uso delle funzioni

- **migliora la leggibilita' e comprensibilita' del sistema che si sta realizzando;**

- **evita di scrivere piu' volte una serie di operazioni che vengono usate spesso e CON UN COMPITO SPECIFICO.**

NON USARE MAI UNA FUNZIONE PER REALIZZARE UNA REGOLA

La sintassi per una generica funzione:

```
(deffunction <function-name>[optional comment]
   (?arg1 ... ?argM [$?argN])
   (<action 1>
   ....
    <action k>)
)
```

- ***?arg1.. ?argM*** **sono M parametri obbligatori della funzione;**

- ***$?argN*** **e' una variabile multicampo che contiene tutti i parametri che seguono i parametri obbligatori; adatta per parametri opzionali;**

- ***<action ...>*** **sono il corpo della funzione;**

- **La funzione torna sempre il risultato dell'ultima azione quindi il risultato di *<action k>*.**

ESEMPIO:

```
(deffacts Quiz
    (domanda cateti 3 4)
    (risposta A 2.0)
    (risposta B 5.0)
    (risposta C 7.0)
)

(deffunction Ipotenusa
    (?a ?b)
    (sqrt(+ (** ?x 2) (** ?y 2)))
)

(defrule RispostaAlQuiz
    (domanda cateti ?x ?y)
    (risposta ?ID ?valore& =(Ipotenusa ?x ?y))
=>
    (printout t “La risposta è la “ ?ID “ con “ ?valore crlf)
)
```

I/O SYSTEM

CLIPS dispone di funzione per l'I/O verso l'utente e verso il Sitema Operativo:

```
(open <file-name> <logical-name>[<mode>])
(close[<logical-name>])
(printout [<logical-name>] <expression>)
(read [<logical-name>])
(readline [<logical-name>])
(format <logical-name> <string-expression> <expression>)
```

<*logical-name*> e' un identificativo che viene assegnato ad una periferica di I/O al momento dell'apertura e permette di fare riferimento senza conoscerene i dettagli. *t* e' logical name predefinito per *stdout.*

READ e READLINE

Questa funzione legge rispettivamente una parola e una linea dallo standard input o da un file se specificato un logical-name valido per un file.

***read* ritorna un *symbol* e legge un solo campo. Il campo e' delimitato con uno spazio**
ESEMPIO (read):

```
(defrule LeggiInput
   (initial-fact)      ;auto start
=>
   (printout t “Inserisci un colore fondamentale: “)
   (assert (colore (read) ))
)
```

```
(defrule CheckInput
   (colore ?colore&rosso|giallo|verde)
=>
   (printout t “Il colore inserito e’ corretto !” crlf)
)
```

readline torna una stringa. Legge fino a quando non trova un a capo

ESEMPIO (readline):

```
(defrule Prova-readline
   (initial-fact)
=>
   (printout t “Inserisci una stringa: ”)
   (bind ?stringa (readline))
   (assert-string (str-cat “(“ ?stringa “)” ))
   (assert (stringa ?stringa))
)
```

FORMAT

La funzione *format* permette all'utente di avere un output formattato su una periferica di I/O.

```
(format <logical-name> <string-expression> <expression>)
```

- ***logical name* e' il nome logico della periferica sulla quale si vuole inviare l'output. (*t* per il monitor)**
- ***string-expression* e' la stringa che si vuole scrivere. La stringa puo' contenere dei caratteri di formato**
- ***expression*: elenco degli argomenti che devono essere formattati in *string-expression***

LA STRINGA DI FORMATTAZIONE: *%-M.Nx*

- **- indica di allineare a sinistra**
- ***M* indica la larghezza totale del campo (compreso punto e decimali);**
- ***N* indica il numero di decimali da utilizzare;**
- ***x* e' un flag che serve a specificare il tipo di dato che si vuole formattare;**

ELENCO DEI FLAG VALIDI

d = long integer	x = unsigned exadecimal int
f = floating point	s = stringa
e = floating point in base 10	n = inserisce una nuova linea
g = generico	r = inserisce un invio
o = numero in notaz. ottale	% = inserisce il carattere %

```
(format t "un intero: <%d>" 12)
(format t "un float : <%5.2f>" 12.01")
```

DEFGLOBAL

Con `defglobal` si possono definire variabili globali che possono essere usate all'interno di tutto l'ambiente CLIPS

Le variabili globali possono essere usate come parte del pattern matching di una regola , ma cambiando il loro valore non viene riattivato il processo di pattern matching.

La funzione `bind` serve per assegnare un il valore ad variabile globale. Usando `bind` su una variabile globale si resetetta la variabile al suo valore di inizzializzazione.

Una variabile globale puo' essere rimossa dalla memoria con la funzione `clear` o la funzione `undefglobal`

Sintassi:

```
(defglobal [<defmodule-name>] <global-assignment>)

<global-assignment> ::= <global-variable>= <expression>
<global-variable>   ::= ?*<symbol>*
```

DEFGLOBAL - Esempio -

```
(defglobal
   ?*x* = 3
   ?*y* = ?*x*
   ?*z* = (+ ?*x* ?*y*)
   ?*q* = (create$ a b c)
)
(deffacts
   (fatto 7)
   (fatto 2)
)
(defrule esempio
   (fatto ?y&:(> ?y ?*x*))
```

```
=>
    (printout t "esempio attivato" crlf))
```

DEFMODULE

CLIPS supporta lo sviluppo e l'esecuzione modulare si sistemi esperti

Un modulo di CLIPS raggruppa una serie di regole, variabili, fatti ecc. in unico blocco il cui accesso e' controllato con esplicite funzioni.

La gestione dei moduli in CLIPS e' gerarchica:
se un modulo A puo' vedere all'interno di un modulo B allora per il modulo B non e' possibile vedere nulla del modulo A

Un modulo e' pensabile come una scatola che rende visibile al suo esterno (agli altri moduli) solo determinati fatti, regole, ecc..

DEFMODULE - Sintassi

```
(defmodule <module-name> [<comment>] <port-spec>)

<port-spec> ::=  (export <port-item>) |
                 (import <module-name> <port-item>)

<port-item> ::=  ?ALL |
                 ?NONE |
                 <port-construct> ?ALL |
                 <port-construct> ?NONE |
                 <port-construct> <construct-name>|

<port-construct> ::= deftemplate |
                     defclass |
                     defglobal |
                     deffunction |
                     defgeneric
```

il campo port-item e export-item definiscono l'elenco dei componenti di un modulo che vengono rispettivamente esportati verso altri moduli ed importatati da qualche modulo in particolare.

Il CLIPS ha un modulo predefinito che si chiama MAIN che non importa ed esporta alcun componente. Il modulo MAIN puo' essere ridefinito.

ESEMPIO:

```
(defmodule Prova
   (import Bar ?ALL)
   (import Yak deftemplate ?ALL)
   (import Goz defglobal x y z)
   (export defgeneric +)
   (export defclass ?ALL)
```

```
)
```

E' possibile assegnare una regola, fatto o altro ad un modulo, direttamente indicando il nome del modulo prima del nome del componete che si sta creando.

```
(defrule DETECTION::FindFault
   (sensor (name ?name) (value bad))
=>
   (assert ( fault (name ?name)))
)
(defglobal DETECTION ?*count* = 0)
(deffacts DETECTION::Sensor
   (sensor (name v1) (value 0))
)
```

Il modulo corrente e' detto *focus* ed è quello che viene utilizzato per il pattern matching, come base dei fatti ecc..

Per default il focus e' il modulo MAIN.

ESEMPIO 1:

definisco il moduli A e B.

```
CLIPS> (defmodule A)
CLIPS> (defmodule B)
```

ora il focus e' il modulo B. La regola foo viene assegnata al modulo B. Se non viene specificato il modulo ogni istanza viene sempre assegnata al focus

```
CLIPS> (defrule foo => (assert (B 1)) )
```

Creo una regola che assegno al modulo A. CLIPS automaticamente sposta prima il focus sul modulo indicato poi genera la regola. In questo modo rimae attivo il focus sul modulo A

```
CLIPS> (derule A::bar => (assert (A 1)) )
```

Ora chiedo l'elenco delle regole per il modulo corrente. Viene indicata la regola bar che e' l'unica del modulo A

```
CLIPS> (list-defrules)
```

```
bar
```

Sposto il focus sul modulo B. Ora vengono considerati solo gli elementi definiti come appartenenti al modulo B.

```
CLIPS> (set-current-module B)
A
```

Infatti:

```
CLIPS> (list-defrules)
foo
```

ESEMPIO 2:

```
(defmodule MAIN (export ?ALL))

(defrule MAIN::Esempio
   (initial-fact)
=>
   (printout t “Scatta la regola dal modulo MAIN” crlf)
   (focus A B)
)
```

```
(defmodule A (import MAIN deftemplate initial-fact) )

(defrule A::Regola1
   (initial-fact)
=>
   (printout t “Scatta la regola dal modulo A” crlf))


(defmodule B (import MAIN deftemplate initial-fact) )

(defrule B::Regola1
   (initial-fact)
=>
   (printout t “Scatta la regola dal modulo B” crlf))
```